HARDWARE C - A LANGUAGE FOR HARDWARE DESIGN

David C. Ku and Giovanni De Micheli

Technical Report No. CSL-TR-88-362

August 1988

Computer Systems Laboratory
Departments of Electrical Engineering and Computer Science
Stanford University
Stanford, CA 94305

N00014-87-K-0828

Abstract

High-level synthesis is the transformation from a behavioral level specification of hardware to a register transfer level description, which may be mapped to a VLSI implementation. The success of high-level synthesis systems is heavily dependent on how effectively the behavioral language captures the *ideas* of the designer in a simple and understandable way. This paper describes **HardwareC**, a hardware description language that is based on the C programming language, extended with notions of concurrent processes, message passing, explicit instantiation of procedures, and templates. The language is used by the HERCULES High-Level Synthesis System.

Key Words and Phrases: High-level synthesis, hardware description languages.

HardwareC – A Language for Hardware Design

David C. Ku Giovanni De Micheli

Computer Systems Laboratory
Stanford University
Stanford, CA 94305

1 Introduction

High-Level synthesis is the transformation from a behavioral level specification of hardware to a register transfer level description, which may then be mapped to a VLSI implementation. The success of high-level synthesis systems is heavily dependent on how effectively the behavioral language captures the ideas of the designer in a simple and understandable way. This paper describes *HardwareC*, a behavioral hardware description language that is used by the HERCULES High-Level Synthesis system [1,2].

The input to HERCULES consists of two sets of specifications – a description of the *functionality* and a set of *design constraints*. The functionality is described in a C-based language extended for hardware description called *HardwareC*. The *design constraints* specify the timing and resource limitations that are imposed on a given design. The HardwareC description is parsed and translated into a parse tree abstraction called the behavioral intermediate form, which is the basis for behavioral synthesis. Behavioral synthesis performs transformations similar to those found in optimizing compilers. Upon completion of behavioral synthesis, the optimized intermediate form is mapped to a register transfer level implementation.

2 Motivations

Many hardware description languages have been proposed and used in both academia and in industry. Most hardware description languages are oriented towards simulation. As high-level synthesis systems mature, a need arises for languages that aid not only in the *simulation* of hardware, but also in its *design* as well.

Several criteria must be met by a language for hardware design, they are described below.

1. *Supports full spectrum of design styles.*

   The language should support readily the varying spectrum of design styles of the designer, ranging from a pure behavioral description that is independent of the structural implementation, to a mixture of behavior and structure, to a pure structural description of the interconnection and instantiation of hardware modules.

   This criterion is crucial in a design environment since very often the designer has a particular structure in mind when designing hardware. This partial structure should be captured by the language, and reflected in the results of synthesis. A design often requires interfacing to an existing hardware unit, such as an ALU or incrementer. The ability to interface with external *structure* is of utmost importance in automated synthesis.

   Many synthesis systems and hardware description languages support only a specific design style, either pure structure or pure behavior. We believe a more effective approach to design is to use a flexible underlying language that captures the *essence* of the design from the designer, whether that essence be behavioral or structural.

2. *Supports simulation.*

   The language should support simulation in order to ascertain the correctness of a given description. As designs become bigger and more complex, it becomes more important to be able to simulate at all levels of synthesis, from behavioral to structural to logic to gate level.

3. *Simple to learn and use.*

   The language is a tool that the designer uses to capture and transform abstract ideas into complete designs. The tool must therefore be simple to learn and easy to use. Specifically, the language should contain the most basic constructs that are needed to describe a design. Details such as timing and delay should be left out of the language.

HardwareC attempts to satisfy the requirements stated above. As its name implies, it is based somewhat on the C programming language. However, several enhancements are made to increase the expressive power of the language, as well as to facilitate hardware description. The major features of HardwareC are described below.

- Notions of concurrent *processes* and *message passing*,
- *Templates* that allow a single description for a group of similar behavior (polymorphism). For example, an adder template describes all adders of any given size,

- *Instantiation* of procedures, similar to instantiating objects in object oriented languages, and
- Explicit *Input/Output* commands that access the ports of a given model.

HardwareC can be linked to the THOR simulation environment, which is also based on a C-like simulation language [4].

3 Modeling Hardware Behavior

Hardware behavior is modeled as a collection of concurrent and interacting processes. Each *process* consists of a hierarchy of *procedures*, and the processes interact and synchronize with each other through the use of *inter-process communication* mechanisms. This model is appropriate since hardware modules are allocated resources which continuously operate on a time varying set of inputs. A process upon completion will automatically restart execution with a new set of inputs.

The concept of processes and inter-process communication is powerful for both hardware and software models. In both domains, it allows the designer to:

1. Specify the parallelism between interacting modules at a high level, and
2. Isolate the communication and synchronization points between the processes in an explicit manner.

As an illustration of the use of processes and inter-process communication, consider the Intel 8251 UART (Figure 1). The UART is modeled as four concurrently executing processes. The *main* process accepts commands from the microprocessor and coordinates the execution of the other processes. The *transmitter* process writes data out on the serial interface, and the two receiver processes, *synchronous_receiver* and *asynchronous_receiver*, reads data from the serial interface. Note that the execution of each process is independent with respect to each other, and is synchronized through the use of inter-process communication. Inter-process communication is discussed further in Section 12.

HardwareC is a hardware description language for *synchronous* digital circuits. This is a reflection of the hardware model assumed by the HERCULES Synthesis system. Therefore, there is the notion of a *control state* that is sometimes used to describe the language. A *control state* is defined as an interval of time that corresponds to a system clock cycle in a synchronous system. When a particular operation is said to take one or more states, it means that the execution of the operation requires one or more clock cycles to complete before other operations that depend on it can begin.

The HardwareC language is described in the sections that follow.

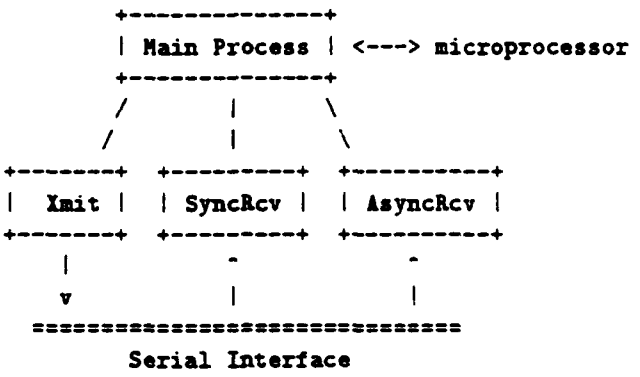

Figure 1: Hardware model for Intel 8251 UART

4 Program Structure

In *HardwareC*, there are two fundamental functional abstraction mechanisms - *process* and *procedure*. A process consists of a hierarchy of procedures, and executes concurrently and independently with respect to the other processes in the system. Similarly, a procedure is also a hierarchy of procedures. However, a procedure executes whenever it is called by another procedure or process.[1]

The transfer of data to and from a process is accomplished through the use of either parameters to the process or through message passing mechanisms (Section 12). The transfer of data to and from a procedure, on the other hand, is accomplished solely through the use of *parameters* to the procedure (Section 11). A procedure can neither return a value as the result of its invocation, nor use message passing to communicate with other procedures. The major differences between a process and a procedure are summarized below.

- *Process.* A process continuously operates on a time-varying set of input data. Upon completion of the last statement in its body, a process will restart its execution, operating on a possibly different set of inputs. An example of the definition of process *procA* is shown below. Note the use of the keyword **process** which prefixes the name of the process.

```
process procA( a, b, c )
      in boolean a;
      out boolean b;
```

[1] No recursive procedures are allowed

```
        inout boolean c[2];
{
        /* body of process */
}
```

- *Procedure*. A procedure can either be combinational or sequential, depending on whether the procedure requires any control states to execute. A sequential procedure begins execution whenever it is *called* by another procedure. Upon completion of execution, a procedure places valid data on its output ports, and returns control to the calling routine. For combinational procedures, execution involves propagating the input data through a network of combinational operations. An example of the definition of procedure *procB* is shown below.

```
procB( x, y, z )
        in boolean x;
        out boolean y;
        inout boolean z[2];
{
        /* body of procedure */
}
```

A procedure cannot be defined within the body of another procedure. This restriction follows the C language, which disallows nested procedural definitions. The resulting flattening of the procedural definition is appropriate since for hardware description, it is more convenient and secure to identify explicitly all inputs and outputs to a given procedure. A procedure defined within the scope of another allows access to all variables that are defined within the scope of its definition. As a result, a procedure's boundary is not well defined if nested procedural definitions are allowed.

Nested procedural definition is different from nested procedural invocation, the latter of which is both permitted and encouraged. For example,

```
/*                              /*
 *  invalid procedure           *  valid procedure
 *    definition                *    definition
 */                             */

procA(a, b)                     validproc(x, y)
   ...                             ...
```

```
{                                   {
    invalidproc(x, y)                   ...
    {                               }
        ...
    }                               procA(a, b)
    ...                                ...
    invalidproc(...)                {
                                        validproc( ... );
}                                   }
```

4.1 Statement Block

Statement block, more commonly known as compound statement, is used to group variable declarations and statements together so that they are syntactically equivalent to a single statement. A statement can either be a variable assignment, an if-then-else statement, a switch statement, a while statement, a for statement, an input/output statement, a message passing primitive, or a block. Semi-colons are used as terminators to statements. A semicolon by itself represents a null statement.

HardwareC supports two types of statement blocks – *parallelizable* blocks and *serial* blocks. Parallelizable blocks are encapsulated using curly braces ({ and }), whereas serial blocks are encapsulated using square brackets ([and]). The differences between the two types are:

- Parallelizable Block { }– The statements within a parallelizable block can all execute in parallel, subject to the data dependencies that exist between the statements. For example,

```
{
        variable_declarations;

        statement1;
        statement2;
}
```

  means that *statement1* can be executed concurrently with *statement2*. The degree of parallelism is determined by the synthesis system.

- Serial Block [ ] – The statements within a serial block are guaranteed to execute in serial order, starting from the first statement in the block. For example, *statement1* will always execute before *statement2*, regardless of their data dependencies.

```
[
        variable_declarations;

        statement1;
        statement2;
]
```

Serial block allows the designer the ability to specify control dependencies between otherwise data independent statements.

A description written using only serial blocks is always guaranteed to be correct – that is, the control dependencies between the statements are fully described. However, the description may not be efficient, since inter-statement parallelism is not exploited. In order to specify such parallelism, the designer should use whenever possible parallelizable blocks ({ }) in describing hardware.

4.2 Parameter Classes

The parameters to processes and procedures are categorized into three different classes: in, out, and inout. Input (in) parameters can only be referenced within the body of a routine; assignments to input parameters are illegal. Output (out) parameters can be modified within the body of a routine; references to output parameters are illegal. Input/output (inout) parameters are bidirectional lines that can be either referenced or assigned. The access protocol to this bidirectional line is left to the designer, and specified as part of the high-level description. Note that an inout parameter is not simply data that will both be read and modified in the routine. It is reserved for the description of bidirectional lines.

For example, *Busy* is an in parameter that controls the access to an inout parameter *Data*. *AllZero* is an output parameter that returns a flag on whether *Data* is all zero.

```
process test( Busy, Data, AllZero )
        in boolean Busy;
        inout boolean Data[8];
        out boolean AllZero;
[
        while ( Busy )
                ;
        /* write to Data */
        Data = newdata;
        write Data;
        AllZero = (Data == 0);
```

]

Notice the use of the serial block ([ ]) to ensure that the write to *Data* occurs after the busy waiting while loop, which does not have any data dependencies with respect to the write.

4.3 Declare Before Use

Whenever a procedure is called, the arguments to the invocation are checked for both compatibility in the variable size and type, as well as for compatibility in the parameter classification. For instance, an input parameter cannot be used as the argument to a procedure call that requires an output parameter. Similarly, an output parameter cannot be used as the argument to a procedure call that requires an input parameter. This compile time consistency checking improves the security of the language.

In order to provide this information to the parser, it is necessary to *declare* a procedure before it can be called. The declaration of a procedure involves specifying:

1. *Name* of the procedure - can be any alphanumeric string beginning with a character.
2. *Number* and *order* of parameters - only Boolean parameters are allowed.
3. *Sizes* of the parameters - the size of a parameter can be specified in terms of a constant, or an expression that evaluates to a constant.
4. *Classes* of the parameters - in, out, or inout.

An example of the declaration for a procedure is shown below.

```
# define MAX 4

declare example( a, b, c )
    in boolean a;
    out boolean b[MAX];
    inout boolean c[MAX+1];
```

The actual names of the parameters are irrelevant; they are used only for the purpose of specifying the classes and sizes of the corresponding parameters. Another example is shown below.

```
declare sum( x, y, z )
    in boolean x;
    out boolean y[2];
```

```
        inout boolean z;

foo( ... )
{
        /* ... */
        sum( a, b, c );
        /* ... */
}
```

If the declaration of *sum* is not supplied, then the subsequent call in `foo` will be invalid. Similarly, for inter-process communication through message passing, it is necessary to predeclare a particular process before sending or receiving messages from it. The declaration for a process is exactly similar to the declaration for a procedure, with the sole exception of the keyword `process` that prefixes the name. For example, the declaration for a process named *foobar* is as follows.

```
declare process foobar( a, b, c )
        in boolean a[3];
        out boolean b[4];
        inout boolean c[2];
```

5 Data in HardwareC

There are two types of *data* entities in the language – constants and variables. They are described in the following sections.

5.1 Constants

There are two types of constants in the language – *integer constants* and *hexadecimal constants*. Integer constants are positive numbers described in the decimal notation. For example, 5 and 223 are integer constants. Hexadecimal constants are numbers described in the hexadecimal notation. They are prefixed by $0x$, followed by a string of hexadecimal digits { 0 – 9, a, b, c, d, e, f }. For example, $0xf$ represents 15, and $0x10$ represents 16. Binary constants are subsets of hexadecimal constants, where 1 is represented as $0x1$, and 0 is represented as $0x0$.

Negative constants are not represented in the language. This restriction stems from the independence of HardwareC to a particular style of complementation. Therefore, if the designer wishes to specify -3 in one's complement notation, then he should specify the bit-wise representation of the value using

hexadecimal constants. For an 8-bit number in one's complement notation, -3 is represented as $0xf8$.

5.2 Variables

There are two variable *types* in the language – *Boolean* and *integer*. Boolean variables are mapped to wires or registers in the final hardware, whereas integer variables are provided for the convenience of the description, and will be resolved at compile time during behavioral synthesis.

A variable may be declared within any block ({ } or [ ]) of any arbitrary nesting. The semantic follows that of block structured languages, where a variable is visible only within the scope of its definition. A variable with the same name at a deeper nesting block level will override any current definition of the variable.

For instance, all declarations in the following example are valid.

```
{
      int i;
      boolean x;
      {
            int i; /* new integer */
            boolean x, y;
            ...
      }
      /* y is not defined here */
}
```

No global variables are allowed in *HardwareC*. This restriction is due to the fact that global variables allow side effects that are not explicitly identified. This is undesirable from the standpoint of security, verifiability, and program readability. If some data must be shared between two routines, then the data should be explicitly specified as common parameters to the two routines.

Integer Integer variables can only be scalar quantities. Integer variables may be used in any arithmetic, Boolean, and relational expressions. They can also be used as indices to constant iteration loops (`for` loop), and as indices for accessing components of Boolean vectors and matrices. The following example demonstrates the use of integer variables and expressions in accessing components of a Boolean vector.

```
/*
 *    swaps the two nibbles in "a" to "b"
 */
```

```
swap(a, b)
    in boolean a[8];
    out boolean b[8];
{
    int i, j, k;

    k = 3;

    /* copies LSB nibble to b */
    for i = 0 to k do
        b[ i+4:i+4 ] = a[ i:i ];

    /* does exactly the same thing */
    i = 0;
    j = 3;
    b[ i+4:j+4 ] = a[ i:j ];

    /* copies MSB nibble to b */
    b[ i:j ] = a[ i+4:j+4 ];

    write b;
}
```

The exact syntax on accessing components of a Boolean vector is discussed in the next section. The example below shows the use of integer expressions and values in control structures.

```
int i;
boolean vec[24];

for i = 0 to 7 do {
    switch (i) {
    case 0:
        vec[ 3*i:3*i+2 ] = 0x7;  /* binary 111 */
        break;
    default:
        vec[ 3*i:3*i+2 ] = i;
        break;
    }
}

/* vec should have the following value:
        111 110 101 100 011 010 001 111
        MSB                         LSB
 */
```

Boolean A Boolean variable represents one or more signals, where each bit of the variable corresponds to a *signal* that can be either 0 or 1. Boolean variables can be scalar, vector, or matrix. For example, the following declarations are all valid Boolean variables. In particular, a is a scalar, b is a vector of five elements starting from index 0, and c is a matrix of 25 elements, with the rows starting from 0 to 4, and columns starting from 0 to 4.

```
boolean a;          /* scalar */
boolean b[5];       /* vector */
boolean c[5][5];    /* matrix */
```

In Boolean vectors, specifying the variable name without brackets, or with empty brackets, represent the *entire* vector. For example, b and b[] are equivalent to b[0:4]. Columns of a Boolean matrix can be accessed similarly. For example, c[2] and c[2][] are equivalent to c[2][0:4]. Since assignments to matrices are not permitted, a reference to c will automatically be converted to c[0][0:4], the first row of the matrix.

For Boolean vectors and matrices, it is also possible to access a *subrange* of values. This is specified by the colon (:) notation. For example, b[2:3] represents a vector of two values that corresponds to the third and fourth element of b. The most significant bit (MSB) is always the higher index, with the least significant bit (LSB) being the smaller index.

Integer variables and expressions can be used in variable declarations to specify the dimensions of the variable, or they can be used to access components and subranges of Boolean variables. For example,

```
int i;

i = 3;
c[i][i:i+1] = b[0:1];
{
    boolean q[i+1];     /* q has 4 elements */
}
```

Boolean variables are further classified as *local*, *static*, and *register*.

- `boolean` – Local Boolean variables are the default. A local boolean is initialized to zero, and its value is not saved across procedure invocations. For example,

  ```
  boolean flag;
  boolean vectorflag[2], matrixflag[2][3];
  ```

- `static` – Static Boolean variables are similar to local Boolean variables, with the semantic difference that their values are retained across procedural invocations. For example,

```
static internal_state[2];
```

Static variables will always be implemented with storage elements such as registers.

- **register** – Register Boolean variables are architected registers that are specified by the designer. Similar to **static** variables, they also retain their values across procedural invocations. Every assignment to a register variable immediately loads the corresponding register with a new value. For example,

```
register status[8];
```

The difference between **register** and **static** variables is in how assignments are handled, which is discussed next.

Assignments to **boolean** and **static** variables are resolved during behavioral synthesis, and hence do not require any control states for run-time execution. In contrast, each assignment to a **register** variable corresponds to the loading of the register with a new value, and hence requires a control state at run-time.

To demonstrate the differences between static and register variables, consider the two examples below. In procedure **foo**, each assignment to the **static** variable *c* will not consume a control state at run-time. This is due to the fact that the assignment only changes subsequent references to *c*, and hence does not imply loading the register that implements *c* with a new value.

```
foo()
{
    static c;

    c = 1;    /* change reference only */
    c = 0;    /* change reference only */
    c = 1;    /* change reference only */
    c = 0;    /* last value of c is 0 */
}
```

Similarly, the **register** variable *c* in procedure **bar** also has a final value of 0. The difference is that during the execution of **bar**, the register is loaded with 4 values, corresponding to each assignment to the variable. Therefore, the procedure requires four control states to execute.

```
bar()
{
    register c;
```

```
    c = 1;    /* register has 1 */
    c = 0;    /* register has 0 */
    c = 1;    /* register has 1 again */
    c = 0;    /* last value of c is 0 */
}
```

In terms of port behavior, `static` and `register` variables perform the same action – retain values across invocations. Architected registers allow the designer explicit control over the contents of the register. They are useful for testability purposes where the designer wishes to check the contents of a particular register during execution. For example, architected registers are often used as a status register in a processor description.

5.3 Variable Declaration

The dimension of a Boolean variable may be specified as either a constant, an integer variable, or an integer expression. For instance, consider the declarations for Boolean variables a and b.

```
{
    int i;

    i = 3;
    {
        boolean a[i];  /* a has 3 elements */
        ...
    }
    i = 8;
    {
        boolean b[i+3];  /* b has 11 elements */
        ...
    }
}
```

In fact, even the dimensions of the parameters can be specified as arbitrary integer expressions. The delayed binding of variable dimension to variable definition greatly increases the expressiveness of the language, and improves the flexibility and adaptability of an input description. An illustration of the use of integer expressions in parameter declaration is shown below.

```
# define MAX 8

declare foo(a, b)
    in boolean a[MAX+1];    /* 9 elements */
```

```
out boolean b[MAX+2];   /* 10 elements */
```

Note that the integer expressions (MAX+1) and (MAX+2) are used to declare the dimensions of the parameters.

6 Control Flow Constructs

HardwareC supports a single-in, single-out control flow, similar to the Pascal programming language. This implies that no gotos and returns are allowed in the language. Such restriction is appropriate since by supporting a single-in, single-out control flow, the semantics of the language is made simpler, which greatly aids in the correctness verification of programs. The four major control flow constructs are *if*, *switch*, *for*, and *while*; they are described below.

- *if*

  selects among two alternatives, depending on whether the conditional expression evaluates to TRUE or FALSE, non-zero or zero, respectively. The conditional expression can be any arithmetic, Boolean, and relational expression that involves both integer and Boolean variables. The else part may be unspecified. For example,

```
if ( ! b & chipselect )
    a = 0;     /* any statement */
else
    a = 1;     /* any statement */
```

- *switch*

  selects among one or more alternatives, depending on the value of the switch conditional expression. The individual cases in the switch statement may be cascaded, and are delimited through the use of break statements.

```
switch ( <switch expr> ) {
case num1:
    <statement>
case num2:
case num3:
    <statement>
    break;
...
default:
    break;
}
```

`break` statements are illegal in any other contexts, such as for premature exits from while loops.

- *for*

  is a constant bound iteration on a given integer variable. The exact syntax is as follows,

  ```
  for <int var> = expr1 {to|downto} expr2 [step expr3]
      do <statement>
  ```

  where expr1, expr2, and expr3 can be any constant or integer expression. The `step` clause is optional, and has a default of one.

- *while*

  is a data dependent iteration on a given Boolean expression. The syntax of the while loop is as follows.

  ```
  while ( loop_expr )
      <statement>
  ```

  loop_expr can be any integer, relational, or Boolean expression.

There are only two types of iterative loop constructs in HardwareC - `for` loops and `while` loops. For loops are deterministic iteration loops, whose bounds are known at compile time. A while loop is a non-deterministic iteration whose exit condition can be data dependent, and hence is in general unknown at compile time. Whereas there are many variants of data-dependent loops, such as `do-while` and `repeat-until`, they can all be written in terms of `while` loops. Including the many variants of data-dependent loops does not increase the expressive power of the language. Therefore, for reasons of simplicity, HardwareC supports only one style of data-dependent loops - the `while` loop.

7 Assignments

When a program references a particular variable at different locations in the code, it may reference different *values*, depending on whether the variable has been re-assigned between the references. The *value* of a variable is defined to be the data most recently assigned to it. [2] An assignment to a variable modifies the *value* of the variable.

Both Boolean and integer variables, as well as inout and out parameters, can be assigned. Note that an assignment is not an expression that returns a

[2] Data is defined to be the results of procedure call, binary and unary operators, or message passing.

value. For example, $a = a + 1$ does not return the new value of a, which is in contrast to the C programming language.

The semantics of assignment to different types of variables are described below.

- **out or inout parameters.**

  Assignments to an output or input/output parameter will update the *value* of the port. The actual reflection of the port values to externally visible signals is performed explicitly through the use of input/output commands, discussed in Section 11.

- **int, boolean, or static variables.**

  Assignments to these variables will be resolved and removed during behavioral synthesis. Therefore, the assignments serve only to update the *value* of the variable for subsequent references, and do not consume any control states at run-time. Only constants or integer expressions can be assigned to integer variables. There is no restriction on the values that can be assigned to Boolean variables.

- **register variables.**

  An assignment to an architected register loads the register with a new value. Therefore, every assignment requires a control state for execution at run-time.

8 Templates

Very often two descriptions differ in only very restricted ways. For example, they are the same with the sole exception that the variable sizes are different, as illustrated below for a four-bit and a five-bit adder.

```
/*
 *  Four-Bit adder
 */
adder4(a, b, c, cin, cout)
    in boolean a[4], b[4], cin;
    out boolean c[4], cout;
{
    /* 4 bits add */
}

/*
 *  Five-Bit adder
 */
```

```
adder5(a, b, c, cin, cout)
    in boolean a[5], b[5], cin;
    out boolean c[5], cout;
{
    /* same as above, but for 5 bits */
}
```

It is much simpler and expressive if only *one* description is given for the adder function which takes an argument specifying the size of the operation. This approach offers the advantages of (1) consistency of descriptions, (2) economy of code, which decreases design time, and (3) reusability of code (polymorphism).

In HardwareC, the mechanism which supports parameterized descriptions is a *template*. A template can either be used to generate a procedure or a process. Templates are similar to generic packages in ADA, or generic classes in several object oriented languages. A template takes one or more integer arguments as parameters, and given a particular mapping of integer values to the integer parameters, a *corresponding instance* can be obtained. A good analogy can be made between templates and module generation; in fact, a template is a form of high-level module generation. The exact syntax of templates for procedures and processes is given below.

- Procedure Template definition:

```
template <procedure_name> ( <parameters> )
                    with ( <integer_parameters> )
    <parameter declarations>
{
    <body>
}
```

- Process Template definition:

```
template process <procedure_name> ( <parameters> )
                              with ( <integer_parameters> )
    <parameter declarations>
{
    <body>
}
```

The keyword **template** prefixes the name of the template, and the keyword **with** separates the Boolean parameters from the integer parameters. The **integer_parameters** are the names of the integer parameters, and are separated by commas (,) if more than one is present. These integer parameters can be

used in both parameter declaration and the body of the template as integer constants. Specifically, assignment to an integer parameter is not allowed.

Let us consider the description of a template for the ripple-carry adder function.

```
/*
 *     ripple carry adder
 */
template adder(a, b, c, cin, cout) with (size)
    in boolean a[size], b[size], cin;
    in boolean c[size], cout;
{
    int i, j;
    boolean temp;

    temp = cin;
    j = size -1;
    for i = 0 to j do {
        c[i:i] = a[i:i] xor b[i:i] xor temp;
        temp = a[i:i] & b[i:i] |
               temp & (a[i:i] | b[i:i]);
    }
    cout = temp;
    write c, cout;
}
```

Templates can be used in two ways, corresponding to the environment level and the language level.

1. *Environment Level* – The HERCULES Synthesis system can create any number of instances of a given template. This is useful for example in generating library units such as adders or incrementers.

2. *Language Level* – Within the description, the designer can make references to particular instances of a template through *instantiating* templates. Template instantiation is described next.

9 Instances

HardwareC supports *explicit instantiation* of procedures and procedure templates in the description. A instance of a procedure represents an *object* that encapsulates both behavior and state. In a similar manner, a Boolean variable is also an *object* whose behavior is specified by the language in terms of the semantics of accessing and modifying the variable. Instances can therefore

be treated as *instance variables* that are declared and used in the scope of its definition. The syntax of procedure instantiation is described below.

```
instance <procedure_name> inst1, inst2, ..., instn;
```

The keyword **instance** prefixes the name of the procedure , followed by the names of the instances to be created, separated by commas. If a template is instantiated, the syntax is described as follows.

```
instance <template_name>( <integer_arg> ) t1,...,tn;
```

the arguments to the integer parameters should be specified, separated by commas. The integer arguments must be constants, and cannot be any variables or expressions. Note that the scoping rules for variable visibility also apply to instances. Consider the example below, where **counter** is a procedure that increments an internal variable each time it is called.

```
{
    instance counter a;
    instance adder(4) o4;     /* 4 bit adder */

    {
        instance counter a, b;

        a(...); /* new counter */

        /* can access o4 also */
    }

    a(...);  /* old counter */

    /* b is undefined here */
}
```

The instance **a** of **counter** is different for each different nesting of the block.

9.1 Calling a Procedure

A procedure may be *called* by another process or procedure. This is accomplished by specifying the name of the procedure to be called, along with the arguments to the procedure separated by commas and enclosed in parentheses. A procedure must be *declared* or *defined* before it can be called, otherwise the call will result in an error.

Valid arguments to a procedure call depend on the particular class of the corresponding parameters. Specifically,

- **In Parameter** – All `in` parameters, `inout` parameters, local `boolean`, `static`, and `register` variables and expressions are allowed to be used as arguments in the procedure call.
- **Out Parameter** – All `out` parameters, `inout` parameters, local `boolean` variables, and `static` variables are allowed. No `register` variables are allowed.
- **Inout Parameter** – Only `inout` parameters are allowed.

There are two types of procedure calls – *generic* or *instantiated* calls. A generic call is a call made to a given procedure *type*. The particular instance of the procedure type that is used to implement the call is not specified. To invoke a particular instance of a procedure, the name of the instance simply replaces the name of the procedure in a procedure call. This style of procedure call is called an *instantiated* procedure call. For example,

```
{
    instance counter x;

    counter(...);     /* generic call */
    x(...);           /* instantiated call */

    counter(...);     /* generic call */
    x(...);           /* instantiated call */
}
```

All the calls to `x` will invoke the same instance. However, if a procedure is invoked without specifying the instance (`generic` procedure calls), then the synthesis system is free to determine whether the call can be shared with other generic calls, or whether to allocate an instance to the call.

9.2 Advantages of Instantiating Procedures

There are several advantages in supporting both generic and instantiated procedure calls. They are briefly described below.

1. *Resolves Ambiguity in the behavior.* The designer can *completely* describe the behavior that is intended without relying on hidden assumptions.
2. *Access to both State and Behavior.* The designer can access not only behavior through procedure calls, but also internal state information as well.
3. *Supports Spectrum of Design descriptions.* Depending on the style of the designer, hardware can be described in a spectrum ranging from *pure*

behavior that is free from structural implications to *pure structure* that describes the interconnection and instantiation of hardware components. A procedure instantiation is similar to instantiating a hardware module. and therefore HardwareC supports fully the spectrum of design description styles.

4. *Specifies Resource Sharing at description level.* Although it is not required by the synthesis system, it is possible for a designer to specify the sharing of resources (procedure instances) through the use of instantiating procedures. For example, the designer can specify whether only one adder should be used to implement a description verses one that uses two adders.

9.3 Motivation and Example

A major drawback with many languages is the inability to specify exactly which instance of a given procedure is invoked in a procedure call. This restriction is reasonable for procedures that describe only the functionality without internal state information. However, if a procedure has internal state associated with it (through the use of either static or register variables), such restriction severly handicaps the usability and expressiveness of the language. In fact. the such deficiency can result in either inefficient or even incorrect implementation. depending on whether the assumptions made by the synthesis system matches those made by the designer.

Consider the description of a counter below.

```
/*
 *    each call to it increments by 1
 */
counter(value)
    out boolean value[8];
{
    static state[8];

    state = state + 1;
    value = state;
    write value;
}
```

Every call to the counter module will increment the corresponding internal state variable by one. If a call is made to counter without specifying the particular instance that is to be invoked, then one of two situations will arise.

1. Single instance assumption – If the synthesis system assumes that one and only one instance is associated with a procedure, then a call to counter

will always increment the same internal state (corresponding to the single instance).

However, this approach is overly restrictive since one of the powers of synthesis systems is to explore the spectrum of design tradeoffs between parallel and serial implementations, and by always assuming one instance per procedure this exploration is not possible.

2. No assumption on the invoked instance – On the other hand, if no assumptions are made on which instance a given call will invoke, the synthesis system will then have the flexibility to either dedicate an instance to the call, or share several procedure calls onto the same instance. However, if the procedure has internal state information, then the description can be incorrect, dependent on the particular mapping of procedure calls to procedure instances.

The assumptions that are made by the synthesis system may not be what the designer had in mind when writing the description. For instance, in the code segment below, counter is called twice.

```
counter(sum1);
...
counter(sum2);
...
```

The designer can either view the two calls as incrementing the same value twice, or he can view the two calls to be distinct, each incrementing a value independent of the other. Through instantiation of procedures, the designer can explicitly specify the exact semantics of a procedure call. For example, if the designer wishes to increment a single value twice, then the corresponding code is given below.

```
{
    instance counter value;

    value(...);     /* increment */
    value(...);     /* increment again */
}
```

On the other hand, if the designer wishes to increment two different values, then the code is as follows.

```
{
    instance counter value1, value2;

    value1(...)     /* increment value1 */
```

```
    value2(...)     /* increment value2 */
    value1(...)     /* increment value1 again */
}
```

The designer can instantiate not only procedures, but also procedure *templates*. This is accomplished by supplying the values to the integer parameters to the corresponding template, separated by commas if more than one value is required. The example below makes use of the adder template described in Section 8.

```
{
    instance adder(4) o4;     /* 4 bit adder */
    instance adder(5) o5;     /* 5 bit adder */

    o4(...);
    o5(...);
}
```

10 Operators

HardwareC supports all Boolean and relational operators available in the conventional C programming language. It also supports all arithmetic operators. The operators can be unary or binary, and take both integer and Boolean variables as operands. Mixed operations between Boolean and integer variables are also allowed if it makes sense. For instance, Boolean inversion on an integer variable is illegal.

The operators are summarized below.

Arithmetic { $-$, $-$, $*$, / } Applies to both integer and Boolean variables and expressions.

Boolean { !, &, |, `xor` } bitwise Boolean operators, shift left (<<), shift right (>>), rotate left (`rl`), and rotate right (`rr`). Applies to only Boolean variables and expressions.

Relational { !=, ==, <=, >=, <, > } Applies to both integer and Boolean variables and expressions.

Auto-Increment/Auto-Decrement { $++$, $--$ } Applies to both integer and Boolean variables and expressions. For example, $a++$ and $++a$ are equivalent to $a = a + 1$, whereas $a--$ and $--a$ are equivalent to $a = a - 1$.

11 Input/Output

HardwareC has explicit input and output commands to allow reading from and writing to the ports of a process or procedure. The three main commands are *write*, *free*, and *read*, and are described below.

write writes the most recently assigned 'value' of an output or inout parameter onto the corresponding ports. Different semantics exist for different types of parameters; they are summarized below.

- No write is specified for an inout or out parameter – any change made to that parameter will not be visible. In the example below, the assignments to the inout parameter a do not affect the value of the port; they only serve to alter the value of a for subsequent references.

```
inout boolean a;
...

a = 1;    /* port unaffected */
a = 0;    /* port unaffected */
a = 1;    /* port unaffected */
...
```

- Singe write to an out parameter – in many situations, the designer wishes to connect an output port directly to the result of a particular operation. There are two advantages for using direct connection. First, it does not waste a state at run-time. Second. direct connection allows external visibility of a particular operation.

  Direct connection is achieved by specifying a single write for an output parameter in the body of the routine. For instance, the output parameter z is connected directly to the output of the adder in the example below.

```
while (run) {
    temp = temp + 1;
    z = temp;
    write z;    /* direct connection */
}
```

Any value written to a port will be retained until either the next write or free statement. In the example below, the out parameter c will generate a pulse on the ports.

```
out boolean c;
```

```
   ...

c = 0
write c;  /* port has 0 */
   ...
c = 1;
write c;  /* port has 1 */
   ...
c = 0
write c;  /* port has 0 again */
```

free sets the corresponding output or inout port to high impedance float value. For both **free** and **write**, the effect of the change on port boundary will take place exactly one cycle after the statement begins execution. Any **write** to a port that has been set to float state will overwrite it with the new value. For example,

```
out boolean d;
...

d = 1
write d;    /* port has 1 */
free d;     /* port has high-Z */
write d;    /* port has 1 again */
```

read samples the corresponding in or inout port into a register, and returns the output of the sampling register. Execution of a **read** statement will take one cycle to complete. For example.

```
y = read( x );
/* y is sampled version of x */
```

12 Inter-Process Communication

There are two paradigms for inter-process communication – *shared medium* and *message passing*. Shared medium communication refers to the transfer of information between modules through a common set of ports. The protocol which governs correct handshaking between the modules is provided by the designer, and is described as an integral part of the high-level description. Message passing communication, on the other hand, utilizes explicit *send* and *receive* operations to synchronize between the two concurrent processes.

Each approach has its advantages and limitations. For example, in communication through shared medium, the performance advantage is offset by an increase in the complexity of the resulting high-level description. Likewise, the conceptual elegance of message passing solves both synchronization and communication in systems, but may result in unacceptable implementation complexity if it is used without restraint.

HardwareC offers both approaches. First, it allows *shared medium* communication through the use of *parameters* to processes or procedures. Second, it allows a synchronous *send-receive* message passing scheme with fixed-size messages. The size of a message represents the number of bits that is communicated between the processes, and may be specified by the designer in the input description. Synchronous message passing provides a simple yet powerful approach to inter-process synchronization and limited data transfer without incurring the cost of message buffering.

12.1 Message Passing Primitives

There are three primitive operations in message passing: *send*, *receive*, and *msgwait*. Only processes can use the message passing primitives. *send* transmits a fixed size message to another process. The current process will wait and synchronize until the corresponding process issues a *receive*, whereupon the transfer of information will take place. For example, `targetprocess` is the receiving process, and `message` is the message to be sent.

```
send( targetprocess, message );
```

receive accepts a message from a given process, and will wait and synchronize until the corresponding process issues a *send*. For example, `sourceprocess` is the sending process, and `buffer` is the message received.

```
receive( sourceprocess, buffer );
```

msgwait is a query that returns a scalar Boolean flag signifying whether the specified process is currently sending to the current process. For example, `Producer` and `Consumer` are two processes that synchronize with each other using message passing.

```
process Producer(...)
{
    /* generate item */
    send( Consumer, item );
}

process Consumer(...)
{
```

```
        if ( msgwait(Producer) )
        {
            receive( Producer, item );
            /* consume item */
        }
        else
            /* producer not ready */
    }
```

There is a system wide *message size*, which is the bandwidth of the communication channel between the processes. The default is 8 bits wide, and it can be changed by specifying the size in the description as follows.

```
/* <num> is new message size */
ipcsize = constant_number;
```

ipcsize is a keyword in the language, and constant_number is a positive integer constant. The message size change must be done **before** any message passing operation takes place, as the parser will check to ensure proper size messages and buffers are used in the send and receive operations. The assignment should not be within the body of any particular process or procedure. and should lie between procedural definitions.

13 Miscellaneous

HardwareC relies on the C preprocessor during parsing to handle macro definition (#define) and file inclusion (#include) facilities. The designer is free to use any C preprocessor commands in the description.

14 Appendix

Four detailed examples of hardware description using HardwareC are described below. The first is a four bit carry look-ahead adder. The second is a counter process that uses the four bit adder. The third is the traffic light controller described in the Mead-Conway book. The final example is the Intel 8251 UART description.

Four-Bit Adder

```
add4bit( a, b, carryin, result, carryout )
    in boolean a[4];
    in boolean b[4];
    in boolean carryin;
    out boolean result[4];
```

```
        out boolean carryout;
{
        int i;
        boolean P[4], G[4], new;
        /*
        * calculate propagate and generate
        */
        for i = 0 to 3 do
                P[i:i] = a[i:i] xor b[i:i];
        for i = 0 to 3 do
                G[i:i] = a[i:i] & b[i:i];
        /*
        * calculate carryout
        */
        carryout = G[3:3] | (P[3:3] & G[2:2])
                | (P[3:3] & P[2:2] & G[1:1])
                | (P[3:3] & P[2:2] & P[1:1] & G[0:0])
                | (P[3:3] & P[2:2] & P[1:1] & P[0:0] & carryin);
        /*
        * calculate sum
        */
        new = carryin;
        for i = 0 to 3 do {
                result[i:i] = P[i:i] xor new;
                new = G[i:i] | ( P[i:i] & new );
        }

        write result, carryout;
}
```

Counter

```
process counter( run, load, updown, data, sum )
    in boolean run,
        load, updown,
        data[4];
    out boolean sum[4];
{
    boolean temp[5];

    while ( run ) {
        if ( load )
            temp = data;
```

```
        else {
            if ( updown )
                add4bit(temp, 1, 0, temp[0:3], temp[4:4]);
            else
                add4bit(temp, 0xf, 0, temp[0:3], temp[4:4]);
        }
        sum = temp[0:3];
        write sum;
    }
}
```

Traffic controller

```
/*
 * Mead/Conway Traffic Light Controller
 */

# define HIWAY_GREEN  0
# define HIWAY_YELLOW 1
# define FARM_GREEN   2
# define FARM_YELLOW  3

# define GREEN  1
# define YELLOW 2
# define RED    3

# define TRUE  1
# define FALSE 0

process traffic ( run, Cars,
                  TimeoutL, TimeoutS,
                  HiWayL, FarmL, StartTimer )
   in boolean run;
   in boolean Cars,
              TimeoutL,
              TimeoutS;
   out boolean HiWayL[2],
              FarmL[2],
              StartTimer;
{
   static state[2];
   boolean newstate[2];

   while ( run ) {
```

```
/* combinational logic
   to determine nextstate
*/

switch (state) {
case HIWAY_GREEN:
   HiWayL = GREEN;
   FarmL = RED;

   if (Cars & TimeoutL) {
      newstate = HIWAY_YELLOW;
      StartTimer = TRUE;
   } else {
      newstate = HIWAY_GREEN;
      StartTimer = FALSE;
   }
   break;

case HIWAY_YELLOW:
   HiWayL = YELLOW;
   FarmL = RED;

   if ( TimeoutS ) {
      newstate = FARM_GREEN;
      StartTimer = TRUE;
   } else {
      newstate = FARM_YELLOW;
      StartTimer = FALSE;
   }
   break;
case FARM_GREEN:
   HiWayL = RED;
   FarmL = GREEN;

   if ( ! Cars | TimeoutL ) {
      newstate = FARM_YELLOW;
      StartTimer = TRUE;
   } else {
      newstate = FARM_GREEN;
      StartTimer = FALSE;
   }
   break;
case FARM_YELLOW:
```

```
            HiWayL = RED;
            FarmL = YELLOW;

            if ( TimeoutS ) {
               newstate = HIWAY_GREEN;
               StartTimer = TRUE;
            } else {
               newstate = FARM_YELLOW;
               StartTimer = FALSE;
            }
            break;
         }

         state = newstate;
         write HiWayL, FarmL, StartTimer;
      }
   }
```

Intel 8251 UART There are four processes - main, xmit, sync_recv, and async_recv. They communicate through send/receive message passing primitives.

```
/*
 *      i8251 UART - HardwareC version
 *
 *      Written by David Ku
 *      Stanford University
 */

# define        DataSize        8

# define        forever         1
# define        wait(f)         while (f)

# define        TRUE            1
# define        FALSE           0

/*
 *      field definition
 */

# define       eh              control[7:7]
# define       ir              control[6:6]
```

```
# define      rts           control[5:5]
# define      er            control[4:4]
# define      sbrk          control[3:3]
# define      rxE           control[2:2]
# define      dtr           control[1:1]
# define      txen          control[0:0]

# define      dsr           status[7:7]
# define      syndet        status[6:6]
# define      fe            status[5:5]
# define      oe            status[4:4]
# define      pe            status[3:3]
# define      txe           status[2:2]
# define      rxrdy         status[1:1]
# define      txrdy         status[0:0]

# define      scs           mode[7:7]
# define      nsbits        mode[6:7]
# define      esd           mode[5:5]
# define      ep            mode[4:4]
# define      pen           mode[3:3]
# define      nbits         mode[1:2]
# define      brate         mode[0:0]

/*
 *      hunt_mode()
 *
 *      searches in synchronous
 *      receive mode for sync chars
 */

hunt_mode( rxd, drdy, sync1, sync2, mode )
      in boolean rxd;
      in boolean drdy;
      in boolean sync1[DataSize],
                 sync2[DataSize],
                 mode[DataSize];
{
      boolean done;
      boolean data[DataSize];
      boolean ncount[3];

      done = FALSE;
```

```
        while ( ! done ) {

                data = 0xff;
                while ( data != sync1 ) [
                        wait ( drdy );
                        data[7:7] = read ( rxd );
                        data = data >> 1;
                        done = TRUE;
                ]

                if ( mode[7:7] == 0 )  {

                        ncount[2:2] = 1;
                        ncount[0:1] = nbits;
                        while ( ncount ) [
                                wait ( drdy );
                                data[7:7] = rxd;
                                data = data >> 1;
                                ncount = ncount - 1;
                        ]

                        done = (data == sync2);
                }
        }
}

/*
 *      xmit - transmit process
 */

declare process i8251(ChipSelect,
                WriteEnable, ReadEnable, ChipData,
                data, valid, sync1, sync2, mode,
                control, status)
        in boolean ChipSelect;
        in boolean WriteEnable;
        in boolean ReadEnable;
        in boolean ChipData;
        inout boolean data[DataSize];
        out boolean valid;
        out boolean sync1[DataSize];
        out boolean sync2[DataSize];
        out boolean mode[DataSize];
        out boolean control[DataSize];
```

```
        in boolean status[DataSize];


    process xmit(cts, txd, xdrdy, valid,
                 mode, status, control,
                 sync1, sync2)
        in boolean cts;
        out boolean txd;
        in boolean xdrdy;
        in boolean valid;
        in boolean sync1[DataSize],
                   sync2[DataSize];
        in boolean mode[DataSize],
                   control[DataSize];
        out boolean status[DataSize];
    [
        int i;
        boolean sync_mode;          /* sync mode */
        boolean sync_flag;
        boolean data_ready;
        boolean par;
        boolean ncount[3];          /* # bits */
        boolean dbuf[DataSize];
        boolean xdata[DataSize];
        boolean okay[DataSize];

        free status;

        /*
         *      initialization - valid
         *      true when sync1/sync2 is ready
         */

        if ( valid ) {

              txd = 1; txe = 0;
              write txd;

              sync_mode = (mode[6:7] == 0);

              /*    wait for enable   */

              wait ( txen & cts );
```

```
/*    check for sbrk    */

if (! sync_mode & sbrk)
[      txd = 0;
       write txd;
       wait ( (!sbrk) | (!txen) | (!cts) );
       txd = 1;
       write txd;
]

/*
 *      wait if in async mode,
 *      or send sync char if sync
 */
txe = 1;
write txe;
free txe;

if ( sync_mode )
{      /* check if message are pending */
       if ( msgwaiting(i8251) )
             receive(i8251, xdata);
       else
       {      if (sync_flag)
                    xdata = sync1;
             else
                    xdata = sync2;
             sync_flag = ! sync_flag;
       }
}
else
       receive(i8251, xdata);


/*
 *      send start bit
 */
if ( ! sync_mode )
[      wait (xdrdy);
       txd = 0;
       write txd;
]

/*
```

```
                *       send data
                */
               ncount[2:2] = 1;
               ncount[0:1] = nbits;
               while ( ncount )
               [       wait (xdrdy);
                       txd = dbuf[0:0];
                       write txd;
                       ncount = ncount - 1;
                       dbuf = dbuf >> 1;
               ]

               /*
                *       send parity bits if required
                */
               if (pen)
               [
                       par = xdata[0:0];
                       for i = 1 to 7 do
                               par = par xor xdata[i:i];

                       if (! ep)
                               par = ! par;
                       wait (xdrdy);
                       txd = par;
                       write txd;
               ]

               /*
                *       send stop bits
                */
               if (! sync_mode)
               [       wait (xdrdy);
                       txd = 1;
                       write txd;
               ]

               write status;
       }
]

/*
 *      rcvr_sync -
 *
```

```
 *      receiver synchronous process
 */

process rcvr_sync(rxd, drdy, valid,
          mode, control, status,
          sync1, sync2)
      in boolean rxd;       /* receive serial */
      in boolean drdy;      /* data ready */
      in boolean valid;
      in boolean mode[DataSize];
      in boolean control[DataSize];
      in boolean sync1[DataSize];
      in boolean sync2[DataSize];
      out boolean status[DataSize];
{
      boolean sync_mode;
      boolean par;
      boolean ncount[3];
      boolean data[DataSize];

      /*
       *      free up line
       */

      free status;

      /*
       *      determine initialization
       */

      if ( valid ) {

            sync_mode = (mode[6:7] == 0);

            if ( sync_mode ) [

                  /*
                   *      wait for mode
                   */
                  if ( eh )
                        hunt_mode(rxd, drdy,
                           sync1, sync2, mode );

                  /*
```

```
                        *       start shifting data in
                        */
                       ncount[2:2] = 1;
                       ncount[0:1] = nbits;
                       while ( ncount ) [
                             wait ( drdy );
                             data[7:7] = read ( rxd );
                             data = data >> 1;
                             ncount = ncount - 1;
                       ]

                       /*
                        *       send data to main process
                        */
                       send( i8251, data );

                 ]

                 write status;
           }
     }


     /*
      *       rcvr_async -
      *
      *       receiver asynchronous process
      */

     process rcvr_async(rxd, drdy, valid,
           mode, control, status)
           in boolean rxd;       /* receive serial data */
           in boolean drdy;       /* data ready */
           in boolean valid;
           in boolean mode[DataSize];
           in boolean control[DataSize];
           out boolean status[DataSize];
     {
           int i;
           boolean sync_mode;
           boolean par;
           boolean ncount[3];
           boolean data[DataSize];
```

```
/*
 *      determine initialization
 */

free status;
if ( valid ) {

      /* assume mode is stable now */

      sync_mode = (mode[6:7] == 0);

      if ( ! sync_mode ) [

            /*
             *      wait for start bit
             */
            wait ( rxd );
            wait ( ! rxd );

            /*
             *      start shifting data in
             */
            ncount[2:2] = 1;
            ncount[0:1] = nbits;
            while ( ncount ) [
                  wait ( drdy );
                  data[7:7] = read ( rxd );
                  data = data >> 1;
                  ncount = ncount - 1;
            ]

            /*
             *      sample parity bit
             */
            if ( pen ) [

                  par = data[0:0];
                  for i = 1 to 7 do
                        par = par xor data[i:i];

                  if ( ep )
                        par = ! par;
                  wait ( drdy );
                  if ( par != rxd ) {
```

```
                        /* parity error */
                        pe = 1;
                        write pe;
                    }
                ]

                /*
                 *      sample stop bit
                 */
                wait ( drdy );
                if ( rxd == 0 ) {
                    /* framing error */
                    fe = 1;
                    write fe;
                }

                /*
                 *      send data to main process
                 */
                send( i8251, data );

            ]

            write status;
        }
    }


    /*
     *      main process for intel 8251
     */

    process i8251(ChipSelect, WriteEnable,
       ReadEnable, ChipData, data, valid,
       sync1, sync2, mode, control, status)
          in boolean ChipSelect;
          in boolean WriteEnable;
          in boolean ReadEnable;
          in boolean ChipData;
          inout boolean data[DataSize];
          out boolean valid;
          out boolean sync1[DataSize];
          out boolean sync2[DataSize];
          out boolean mode[DataSize];
```

```
        out boolean control[DataSize];
        in boolean status[DataSize];
[
        boolean modebuf[DataSize];
        boolean dbuf[DataSize];
        boolean decode[3];
        boolean sync_mode;

        valid = 0;
        write valid;

        /*
         *      reset sequence: read mode character
         */
        wait ( ChipSelect & WriteEnable & ChipData );

        modebuf = read ( data );

        mode = modebuf;
        valid = 1;

        sync_mode = (modebuf[6:7] == 0);

        /*
         *      read sync characters if necessary
         */

        sync1 = 0;
        sync2 = 0;

        if ( sync_mode )
        [       /* read first sync char */
                wait (ChipSelect&WriteEnable&ChipData);
                sync1 = read ( data );

                /* read second sync char */
                if ( ! modebuf[7:7] )
                [       wait (ChipSelect&WriteEnable&ChipData);
                        sync2 = read ( data );
                ]
        ]

        /* write to output port */
        write valid, mode, sync1, sync2;
```

```
/*
 *      main interp loop
 */

decode[0:0] = ChipData;
decode[1:1] = ReadEnable;
decode[2:2] = WriteEnable;

while ( ChipSelect )
{     switch (decode) {
      case 0x2:          /* read data */
            [
            if (sync_mode)
                  receive(rcvr_sync, dbuf);
            else
                  receive(rcvr_async, dbuf);

            wait ( ! WriteEnable );
            data = dbuf;
            write data;
            ]
            break;
      case 0x3:          /* read status */
            [
            data = read ( status );
            wait ( ! WriteEnable );
            write data;
            ]
            break;
      case 0xC:          /* write data */
            dbuf = read ( data );
            send(xmit, dbuf);
            break;
      case 0xD:          /* write control */
            dbuf = read ( data );
            control = dbuf;
            write control;
            break;
      }
}

]
```

References

[1] David C. Ku, G. De Micheli, *HERCULES - A System for High-Level Synthesis* Proceedings of the 25^{th} ACM/IEEE Design Automation Conference, Anaheim, 1988.

[2] David C. Ku, G. De Micheli, *Using the HERCULES High-Level Synthesis System* Internal report, 1988

[3] Frederic Mailhot, G. De Micheli, *Structural/Logic Intermediate Form Specification* Internal report, 1988

[4] Robert Alverson, Tom Blank, et. al., *THOR User's Manual: Tutorial and Commands* Stanford Technical Report CSL-TR-88-348, January. 1988